Model-Based Testing with Graph Grammars

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

June 8, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process. We describe a model-based testing setup with Graph Grammars. The result is a system for automatic test generation from Graph Grammars. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system is validated using the results of several case-studies.

Contents

1 Related Concepts **2**
- 1.1 Introduction . . . 3
  - 1.1.1 Model-based Testing . . . 3
  - 1.1.2 Graph Transformation . . . 3
  - 1.1.3 Roadmap . . . 4
- 1.2 Research Goals . . . 4
- 1.3 Model-based Testing . . . 4
  - 1.3.1 Previous work . . . 5
  - 1.3.2 Labelled Transition Systems . . . 6
  - 1.3.3 Input-Output Transition Systems . . . 6
  - 1.3.4 Coverage . . . 6
- 1.4 Algebra . . . 7
- 1.5 Symbolic Transition Systems . . . 7
  - 1.5.1 Previous work . . . 8
  - 1.5.2 Definition . . . 8
  - 1.5.3 Input-Output Symbolic Transition Systems . . . 8
  - 1.5.4 Example . . . 9
  - 1.5.5 STS to LTS mapping . . . 9
  - 1.5.6 Coverage . . . 9
- 1.6 Graph Grammars . . . 10
  - 1.6.1 Graphs and morphisms . . . 10
  - 1.6.2 Host graphs . . . 11
  - 1.6.3 Graph transformation rules . . . 11
  - 1.6.4 Graph Transition Systems . . . 11
  - 1.6.5 Example . . . 11
  - 1.6.6 Input-Output GGs . . . 12
- 1.7 Tooling . . . 12
  - 1.7.1 ATM . . . 12
  - 1.7.2 GROOVE . . . 13
  - 1.7.3 Graph grammars in GROOVE . . . 13

2 Design **16**

3 Graph Grammar to STS Algorithm **17**

4 Validation **18**

5 Conclusion **19**

Chapter 1

Related Concepts

1.1 Introduction

In software development projects, often limited time and resources are available for testing. However, testing is an important part of software development, because it decreases future maintainance costs [14]. Testing is a complex process and should be done often [18]. Therefore, the testing process should be as efficient as possible in order to save resources.

citations needed.'this contrasts often-heard statement on testing takes x% of total development cost'

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of test-cases. However, when the creation of a test suite is done manually, this still leaves room for human error [11]. Also, manual creation of test-cases is not time-efficient.

1.1.1 Model-based Testing

Creating an abstract representation or a *model* of the system is a way to tackle these problems. What is meant by a model in this report, is the description of the behavior of a system. UML models are not considered here. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [13]. If the tester copies an error in the document or makes a wrong interpretation, the constructed model becomes incorrect.

A: That's not an argument in favor V: It is in favor of graph grammars vs other model formalisms in model-based testing.

Tools for automatic test generation already exist. One such tool is TorX [24], which integrates automatic test generation, test execution, and test analysis. A version of this tool written in Java under continuous development is JTorX [2]. The testing tool developed by Axini[1] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool will be referred to in this report as Axini Test Manager (ATM).

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency. The feedback process of the stakeholders is obstructed by low transparency models. Models that are often used are state machines, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem in such models with a large number of states is the decrease of model transparency. Errors in models with a low transparency are not easily detected.

1.1.2 Graph Transformation

A formalism with more model transparency is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling than traditional state machines. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]:

> Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship

[1] http://www.axini.nl/

> diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples.

An informative paper on graph transformations is written by Heckel et al. [7]. A quote from this paper:

> Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular.

The graph transformation tool GROOVE[2] is used to model and explore graph grammars.

1.1.3 Roadmap

The motivation above is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect models. The to be designed system, once implemented and validated, provides a valuable contribution to the testing paradigm. The tools GROOVE and ATM are used to create this system.

The structure of the rest of this chapter is as follows: the goals of this research are set out and clarified in section 1.2. The general model-based testing process is set out in section 1.3. Some basic concepts from algebra are described in section 1.4. The symbolic models from ATM are then described in section 1.5. Section 1.6 describes the graph grammar formalism. GROOVE and ATM are described in section 1.7.

1.2 Research Goals

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

1.3 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 1.1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and

[2] http://sourceforge.net/projects/groove/

the responses. It gives a 'pass' or 'fail' verdict depending on whether the SUT conforms to the specification or not respectively.

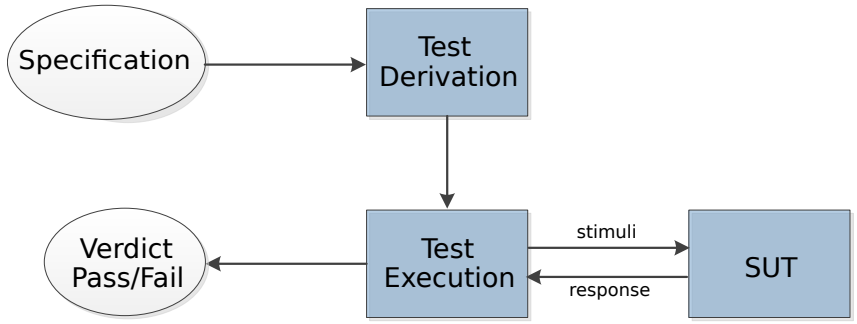


Figure 1.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on-the-fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 1.2. An example of an on-the-fly testing is TorX [24].

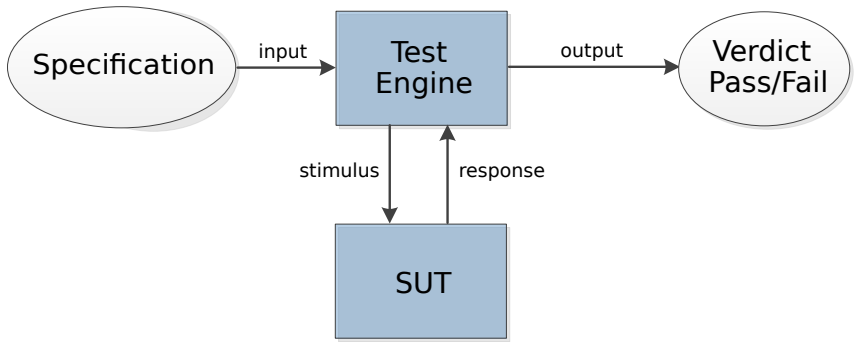


Figure 1.2: A general 'on-the-fly' model-based testing setup

A: Shrink font? V: Why?

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data.

The stucture of the rest of this section is as follows. First, previous work on model-based testing is given. Then, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Finally, the notion of *coverage* is explained.

1.3.1 Previous work

Formal testing theory was introduced by De Nicola et al. [17]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [25] and an algorithm for deriving a sound and exhaustive test suite from a specification in [26]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [12].

1.3.2 Labelled Transition Systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 1.3.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q_0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q_0 \in Q$ is the initial state.

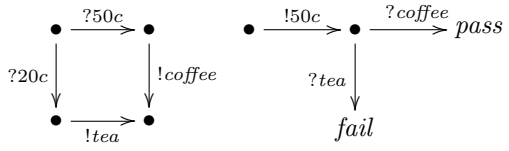
We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

1.3.3 Input-Output Transition Systems

A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [26]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $t \in T$, where $T = \{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test.

An example of such an IOTS is shown in Figure 1.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the outputs are preceded by an exclamation mark. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states. A test case for the coffee machine is given in Figure 1.3b. The test case shows that when an input of '50c' is done, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 1.3: The specification of a coffee machine and a test case as an IOTS

1.3.4 Coverage

The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time

spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [10, 16, 6].

As an example, let us calculate the coverage metrics of the IOTS test case example in 1.3b. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [27]. These statisics are therefore useful metrics for communicating how much of a system is tested.

1.4 Algebra

Some basic concepts from algebra are described here. For a general introduction into logic we refer to [8].

A *multi-sorted signature* $\langle S, F \rangle$ describes the function symbols and sorts of a formal language. F is a set of function symbols, e.g. '+', '*', '=', '<', '0', '1'. S is a set of sorts. Each $f \in F$ has an arity $n \in \mathbb{N}$, where a function symbol with arity $n = 0$ is called a constant symbol. The sort of a function symbol $f \in F$ with arity n is given by $\sigma(f) = s_1 ... s_n + 1$, with $s_i \in S$ for $1 \leq i \leq n$. S_{n+1} is the return sort.

An *algebra* $\mathcal{A} = \langle \mathbb{U}, \mathcal{F} \rangle$ has a non-empty set $\mathbb{U}$ of constants called a *universe*, partitioned into $\mathcal{U}^s$ for each $s \in S$, and a set $\mathcal{F}$ of functions. A function $f_\mathcal{A}$ is typed $\mathbb{U}_\mathcal{A}^{s_1} \times ... \mathbb{U}_\mathcal{A}^{s_n} \rightarrow \mathbb{U}_\mathcal{A}^{s_{n+1}}$, where $s_1 ... s_{n+1}$ is the sort of the function symbol given by the signature. For example, $<_\mathcal{A}$: $\mathbb{U}_\mathcal{A}^{int} \times \mathbb{U}_\mathcal{A}^{int} \rightarrow \mathbb{U}_\mathcal{A}^{bool}$ represents the 'less-than' comparison of two integers.

We define $\mathcal{V} = \mathcal{V}^{int} \uplus \mathcal{V}^{real} \uplus \mathcal{V}^{bool} \uplus \mathcal{V}^{string}$ to be the set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from function symbols F and variables $V \subseteq \mathcal{V}$. The definition of a term is:

$$\begin{array}{rcll} t & ::= & f(t_1 ... t_n) & \\ & | & x & , \textit{ where } x \textit{ is a constant.} \end{array}$$

We write $var(t)$ to denote the set of variables appearing in a term $t \in \mathcal{T}(V)$. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms. An example of a term t is $(x + y)$, with $var(t) = \{x, y\}$. The type of a term is given by:

$$\begin{array}{rll} \sigma : t \mapsto & s & \textit{if } t = x \in \mathcal{V}^S \\ & s_{n+1} & \textit{if } t = f(t_1 ... t_n) \textit{ and } \sigma(f) = s_1 ... s_{n+1}, \textit{ provided } \sigma(t_i) = s_i \end{array}$$

A term with type $\mathbb{U}^{bool}$, is denoted as $\mathcal{R}(\mathcal{V})$. An example is $(x < y)$, where the result is *true* or *false*.

A *term-mapping* is a function $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{V})$. A *valuation* ν is a function $\nu : \mathcal{V} \rightarrow \mathbb{U}$ that assigns constants to variables. For example, given an algebra, $\nu : \{(x \mapsto 1), (y \mapsto 2))\}$ assigns the constants 1 and 2 to the variables x and y respectively. A valuation of a term given $\mathcal{A}$ is defined by $\nu(f(t_1 ... t_n)) \mapsto f_\mathcal{A}(\nu(t_1) ... \nu(t_n))$.

1.5 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. These systems are used in practice in ATM and will therefore be part of GRATiS. In this section, previous work on STSs is given. The definitions of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the transformation of an STS to an LTS is explained and illustrated

by an example. This transformation is useful when comparing STSs to systems that are not STSs. Finally, different coverage metrics on STSs are explained.

1.5.1 Previous work

STSs are introduced by Frantzen et al. [9]. This paper includes a detailed definition, on which the definition in section 1.5.2 is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [22]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [9]. However, the definition of IOSTSs we will use in this report is based on the STSs by [9]. A tool that generates tests based on symbolic specifications is the STG tool, described in Clarke et al. [4].

1.5.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term $t \in \mathcal{R}(\mathcal{V})$. The guard disallows using the switch relation when the valuation of the term results in *false*. When the valuation results in *true*, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the valuation of the term.

Definition 1.5.1. A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, \rightarrow \rangle$, where:

- L is a finite set of locations and $l_0 \in L$ is the initial location.
- $\mathcal{L} \subseteq \mathcal{V}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \rightarrow \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I} \subseteq \mathcal{V}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{R}(\mathcal{V} \cup \mathcal{I}) \times (\mathcal{L} \rightarrow \mathcal{T}(\mathcal{L} \cup \mathcal{I})) \times L$, is the switch relation. We write $l \xrightarrow{\lambda, \phi, \rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the guard and ρ as the update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{L} \cup param(\lambda)$. We define $out(l) \subset \rightarrow$ to be the outgoing switch relations from location l.

1.5.3 Input-Output Symbolic Transition Systems

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type $t \in T$, where $T = \{input, output\}$. As with IOSTSs, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively.

1.5.4 Example

In Figure 1.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The defining tuple of the IOSTS is:

$\langle \{t, m\}, t, \{T, P1, P2, D\}, \{T \mapsto 0, P1 \mapsto 0, P2 \mapsto 2, D \mapsto 0\}, \{d, p, l\}, \{?throw, !move\},$
$\{t \xrightarrow{?throw, 1<=d<=6, D \mapsto d} m, m \xrightarrow{!move, T=1 \wedge l=(P1+D)\%4, P1 \mapsto l, T \mapsto 2} t,$
$m \xrightarrow{!move, T=2 \wedge l=(P2+D)\%4, P2 \mapsto l, T \mapsto 1} t\}\rangle$

The variables $T, P1, P2$ and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $?throw$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throw$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 1.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

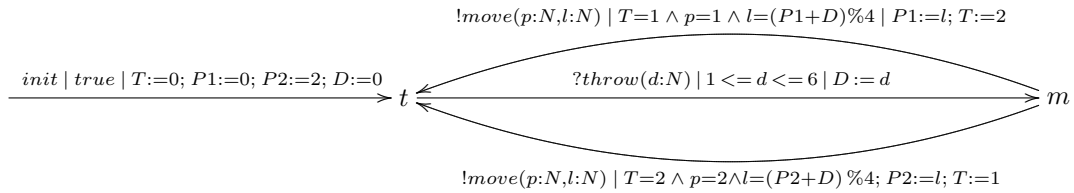


Figure 1.4: The STS of a board game example

1.5.5 STS to LTS mapping

Consider an STS s and an LTS l. There exists a mapping from the location and location variable valuations to the states of l and from the switch relations and variable valuations of s to the transitions of l, such that l is equivalent to s. These relations defined as follows:

Is equivalent the correct term here?

- $\mu_l : (L, \nu : \mathcal{L}) \rightarrow Q$
- $\mu_r : (\rightarrow, \nu : \mathcal{I} \cup \mathcal{L}) \rightarrow T$

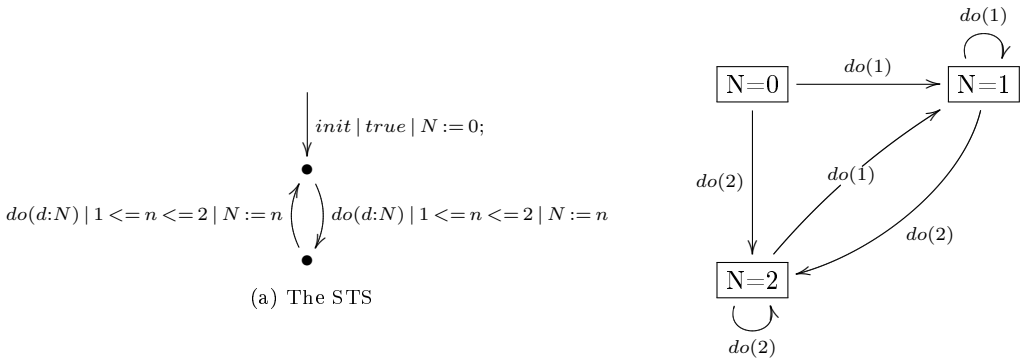
When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ and the number of locations in an STS is considered to be finite, the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 1.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The transformation of a switch relation and concrete values to a transition is also called *instantiating* the switch relation. Another term we will use for a switch relation with a set of concrete data values is an *instantiated switch relation*.

Wat is hier mis mee?

1.5.6 Coverage

The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring



(a) The STS

(b) The LTS

Figure 1.5: An example of a transformation of an STS to an LTS

state and transition coverage of an STS is possible using the LTS resulting from the STS transformation. However, this metric is not always useful, because the number of states and transitions in the LTS depend on the number of unique combinations of concrete values of the variables in the STS. This is potentially very large. For example, when the guards of the switch relations in Figure 1.5a are removed, the transformation leads to an LTS with a state and transition for each possible value of an integer. It is often infeasable to test every data value in the STS. The most interesting data values to test can be found by *boundary-value analysis* and *equivalence partitioning*. For an explanation of these terms we refer to [15]. Boundary-value analysis was found to be most effective by Reid [19] in fault detection.

A:Explain! (example) V: Ik denk eigenlijk niet dat deze technieken relevant gaan zijn uiteindelijk

Data coverage expresses the percentage of data tested in the test run, considering data to be similar if located in the same partition and a better representative of the partition if located close to the partition boundary. These properties of the tested data affect the data coverage percentage.

1.6 Graph Grammars

A *Graph Grammar* (GG) is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.

The rest of this section is ordered as follows: first, host graphs are explained. This is then used to explain graph transformation rules. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, a method for transforming a GTS to an STS is given. For a more detailed overview of GGs, we refer to [20, 7, 1].

1.6.1 Graphs and morphisms

A graph $\mathcal{G}^{graph}$ is a tuple $\langle V, E \rangle$, where:

- V is a set of nodes
- E is a set of edges, where each $e \in E$ has a label and source and target nodes in N

A graph H has an *occurrence* in a graph G, denoted by $H \to G$, if there is a mapping from the nodes and the edges of H to the nodes and the edges of G respectively. Such a mapping is called a *morphism*. A node or edge x in graph H is then said to have an *image* in graph G and x is a

pre-image of the image. A graph H has a partial morphism to a graph G if there are elements in H without an image in G.

1.6.2 Host graphs

A host graph, given an algebra $\mathcal{A}$, is given by: $\mathcal{G}_{\mathcal{A}}^{host} \subseteq \mathcal{G}^{graph} \uplus \mathbb{U}_{\mathcal{A}}^{int} \uplus \mathbb{U}_{\mathcal{A}}^{real} \uplus \mathbb{U}_{\mathcal{A}}^{bool} \uplus \mathbb{U}_{\mathcal{A}}^{string}$.

1.6.3 Graph transformation rules

A rule graph is given by $\mathcal{G}^{rule} \subseteq \mathcal{G}^{graph} \uplus F_{constant}$. A variable $v \in \mathcal{V}^s, s \in S$ can have an image i in a host graph if $i \in \mathbb{U}_{\mathcal{A}}^s$. A constant symbol $c \notin \mathcal{V}$ can have an image i in a host graph if $i \in \mathbb{U}_{\mathcal{A}}^s$ and i is represented by the constant symbol c in the signature.

Definition 1.6.1. A transformation rule is a tuple $\langle LHS, NAC, RHS \rangle$, where:

- *LHS* is a rule graph representing the left-hand side of the rule
- *NAC* is a set of rule graphs representing the negative application conditions
- *RHS* is a rule graph representing the right-hand side of the rule

A rule R is applicable on a graph G if its *LHS* has an occurrence in G and $\nexists n \in NAC$ such that n has an occurence in G and $\forall e \in LHS$, if e has an image i in n, and an image j in G, then j should be an image of i.

This 'applicability' as defined here is refered to as a rule *match*. After the rule match is applied to the graph, all elements in *LHS* that do not have an image in *RHS*, are removed from G and all elements in *RHS* that do not have a pre-image in *LHS*, are added to G.

1.6.4 Graph Transition Systems

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of *graph states* connected by *rule transitions*.

Definition 1.6.2. A graph transition system is an 8-tuple $\langle G, R, L, T_{gts}, g_0 \rangle$, where:

- G is a set of host graphs
- R is a set of transformation rules
- L is a finite set of labels
- $T_{gts} \in G \times (L \cup \{\tau\}) \times G \times R$, with $\tau \notin L$, is the rule transition relation
- $g_0 \in G$ is the initial graph

A: what are the labels? V: hoe bedoel je? strings?

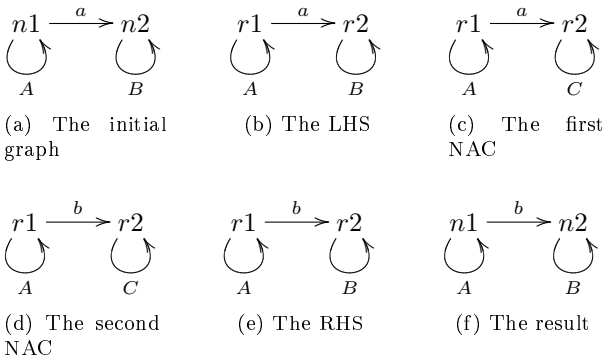
We write $s \xrightarrow{\mu} s'$ if there is a rule transition labelled μ from state s to state s', i.e., $(s, \mu, s') \in T$.

These systems are very similar to LTSs. A GTS can be transformed to an LTS by omitting the graphs, rules and mappings.

1.6.5 Example

Figure 1.6 shows an example of the initial graph and one rule of a GG. The initial graph can be represented by $\langle \{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The *LHS* of the rule has an occurrence in the initial graph. None of the graphs in the *NAC* have an occurrence in the initial

graph, because the edge with label C does not exist in the initial graph. The new graph after applying the rule is in Figure 1.6f.



(a) The initial graph (b) The LHS (c) The first NAC

(d) The second NAC (e) The RHS (f) The result

Figure 1.6: An example of a GG

This picture needs to be placed in a structure, need a way of nesting graphs...

1.6.6 Input-Output GGs

In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule transition label $l \in L$ has a type $t \in T$, where $T = \{input, output\}$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS).

1.7 Tooling

1.7.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 1.7. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 1.2.

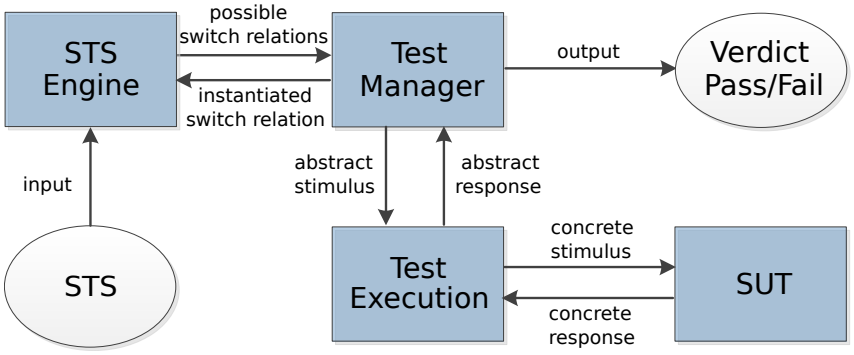


Figure 1.7: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.
2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.
3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label '!ok'. The Test Manager is notified with this abstract response.
5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

1.7.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [21]. It has been applied to several case studies, such as model transformations and security and leader election protocols [5].

The architecture of the GROOVE tool is shown graphically in Figure 1.8. A graph grammar is given as input to the Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTS. The graph states and rule transitions in this GTS can then be inspected using the GUI.

1.7.3 Graph grammars in GROOVE

The running example from Figure 1.4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 1.9. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 1.9b, 1.9c and 1.9d show three rules. Figure 1.9a shows the start graph of the system.

The colors on the nodes and edges in the rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.

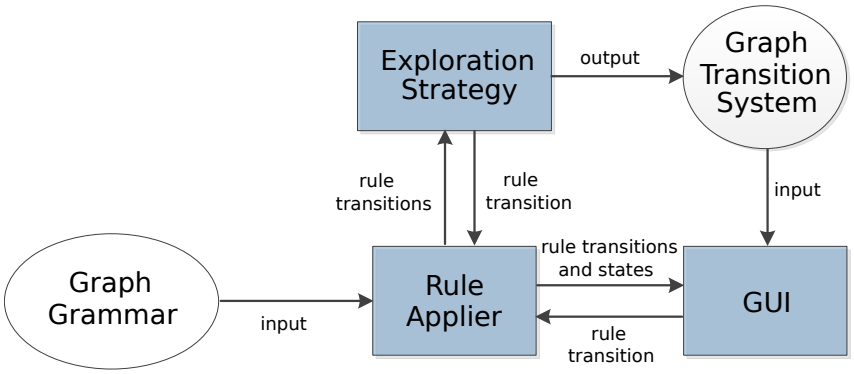
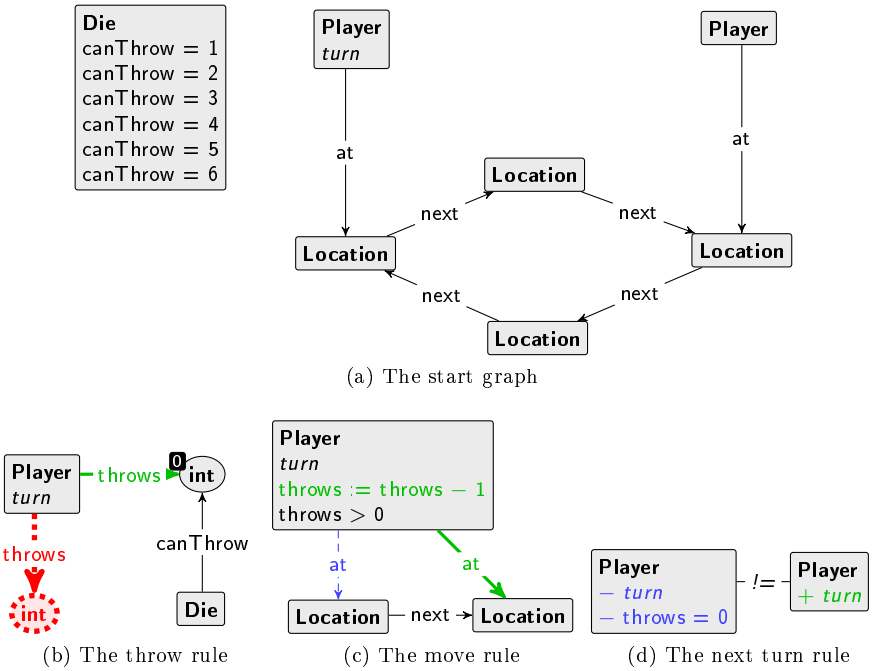


Figure 1.8: The GROOVE Tool



(a) The start graph

(b) The throw rule

(c) The move rule

(d) The next turn rule

Figure 1.9: The graph grammar of the board game example in Figure 1.4

3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The rules can be described as follows:

1. 1.9b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 1.9c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 1.9d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The strings on the nodes are a short-hand notation. The bold strings, **Die**, **Player**, **Location** and **int** indicate the *type* of the node. Nodes with a type starting with a lower case letter, such as **int**, are variable nodes from $\mathcal{V}$. The italic string *turn*, is a representation of a self-edge with label *turn*. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node

and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.

The assignments on the **Die** node are representations of edges labelled 'canThrow' to variable nodes. The six variable nodes are of the type integer and each have an initial value of one to six. The throws value assignment (:=) in the move rule is a shorthand for two edges: one edge in the *LHS* with label *throws* from the **Player** node to an integer node with value i and another edge in the *RHS* with label *throws* from the **Player** node to an integer node with value $i - 1$.

The 'throws $>$ 0' is a term over the variable node that is the target of an outgoing edge labeled 'throws'. In this case, the valuation of the term be true for the rule to match the graph.

The number '0' in the top left of the **int** node in the throw rule indicates that this integer is the first parameter in $param(l)$, where l is the label on the rule transition created by applying the throws rule.

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 1.10 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 1.9a. Figure 1.11 shows the graph represented by s_2.

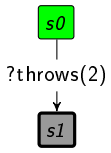


Figure 1.10: The GTS after one rule application on the board game example in Figure 1.9

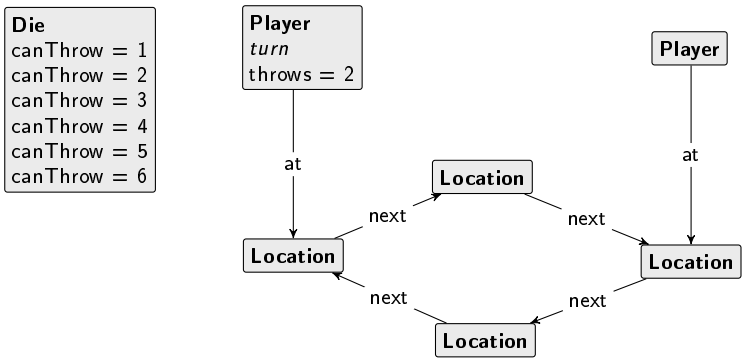


Figure 1.11: The graph of state $s2$ in Figure 1.10

Chapter 2

Design

This chapter covers the design issues of GRATiS. GRATiS transforms a graph grammar to an STS and starts the model-based testing on that STS. This design choice was made, because STSs are practical for model-based testing; variables give support for modelling data values in systems and using STSs supports the separation of state/transition coverage in location/switch relation coverage and data coverage.

Section ?? gives a formal approach of transforming a graph grammar to an STS. Section ?? gives possible optimizations. Section ?? shows the design of GRATiS and elaborates on the choices made. Section ?? gives implementation problems and solutions on GROOVE and ATM.

Chapter 3

Graph Grammar to STS Algorithm

Chapter 4

Validation

This chapter covers the validation of the design. The validation is done through case-studies, reported in section **??**. Measurements on models and the performance are reported in section **??**.

Chapter 5

Conclusion

ACKNOWLEDGEMENTS

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[6] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[7] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[8] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[11] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[12] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[13] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[14] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[15] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[16] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[17] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[18] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap).* Uitgeverij Tutein Nolthenius, 1995.

[19] S.C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64 –73, nov 1997.

[20] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[21] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[22] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[23] Floor Sietsma. A case study in formal testing and an algorithm for automatic test case generation with symbolic transition systems. Master's thesis, Universiteit van Amsterdam, 2009.

[24] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[25] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[26] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[27] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.